SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) LEVEL II

(12)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFOSR-TR- 81-0663	2. GOVT ACCESSION NO. 3. RECIPIENT'S CATALOG NUMBER AD-A105 527
4. TITLE (and Subtitle) A FUNCTIONAL TECHNIQUE FOR DECOMPOSING THE COMPLEXITY OF REQUIREMENTS ANALYSIS.	5. TYPE OF REPORT & PERIOD COVERED TECHNICAL
	6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Pamela Zave	8. CONTRACT OR GRANT NUMBER(s) F49620-80-C-0001
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Computer Science University of Maryland College Park MD 20742	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS PE61102F, 2304/A2
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Office of Scientific Research/NM Bolling AFB DC 20332	12. REPORT DATE AUGUST 1981
	13. NUMBER OF PAGES 21
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DTIC ELECTE OCT 0 9 1981 S D E

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper is based on an 'operational' approach to requirements specification for embedded systems, in which a requirements specification is an executable model of the proposed system interacting with its environment. The operational model is described in terms of asynchronously interacting digital processes. This paper addresses directly the question: How can such specifications for complex systems be developed?
(CONT)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

401963

ITEM #20, CONT:

The proposed technique exploits an identification of intuitively recognizable system functions with processes in the formal specification of that system, so that its process structure can be determined during early analysis. Since the formal language supports incremental development of a specification, one process at a time, the requirements analyst can then elaborate the requirements one function at a time. Since the elaboration of each function entails quite a number of decisions, significant decomposition of complexity is achieved.

The technique is illustrated within the domain of process-control systems, but its extensibility to other applications is argued. The specifications produced are good with respect to several important criteria, including modifiability and representation of performance constraints.

AFOSR-TR- 81 -0663

A FUNCTIONAL TECHNIQUE FOR DECOMPOSING THE COMPLEXITY OF REQUIREMENTS ANALYSIS*

Pamela Zave
Department of Computer Science
University of Maryland
College Park, Maryland 20742
(301) 454-4251

*This work was supported in part by the Air Force Office of Scientific Research under Contract F49620-80-C-0001.

A FUNCTIONAL TECHNIQUE FOR DECOMPOSING THE COMPLEXITY OF REQUIREMENTS ANALYSIS

Abstract

This paper is based on an "operational" approach to requirements specification for embedded systems, in which a requirements specification is an executable model of the proposed system interacting with its environment. The operational model is described in terms of asynchronously interacting digital processes. This paper addresses directly the question: How can such specifications for complex systems be developed?

The proposed technique exploits an identification of intuitively recognizable system functions with processes in the formal specification of that system, so that its process structure can be determined during early analysis. Since the formal language supports incremental development of a specification, one process at a time, the requirements analyst can then elaborate the requirements one function at a time. Since the elaboration of each function entails quite a number of decisions, significant decomposition of complexity is achieved.

The technique is illustrated within the domain of process-control systems, but its extensibility to other applications is argued. The specifications produced are good with respect to several important criteria, including modifiability and representation of performance constraints.

1. INTRODUCTION

Progress in software engineering has always been bottom-up: we understand a thing in detail before we learn to generalize it, abstract from it, and ultimately develop it by generating an abstraction first. The requirements phase of system development is no exception. In the absence of solid, formal techniques for specifying requirements it has been difficult to make precise statements about how requirements can be analyzed and specifications developed.

This paper is an outgrowth of work on specification of requirements for "embedded" (roughly equivalent to "real-time") systems, which has yielded a promising approach embodied in a formal specification language (these results are summarized in 1.1). This paper addresses directly the question: How can such specifications for complex systems be developed? Although we base our arguments partially on the rigors of the target notation, the happy ending is an analysis technique which is intuitively appealing and user-oriented.

1.1. An Operational Approach to Requirements Specification for Embedded Systems

The term "operational" is used to describe our approach because in it a requirements specification is a working model of the proposed system interacting with its environment (Figure 1). The approach has four major characteristics:

(a) There is an explicit model of the environment. This model may be of great assistance in requirements analysis (and communication with the user), since the purpose of any system is to support a desired mode of operation in

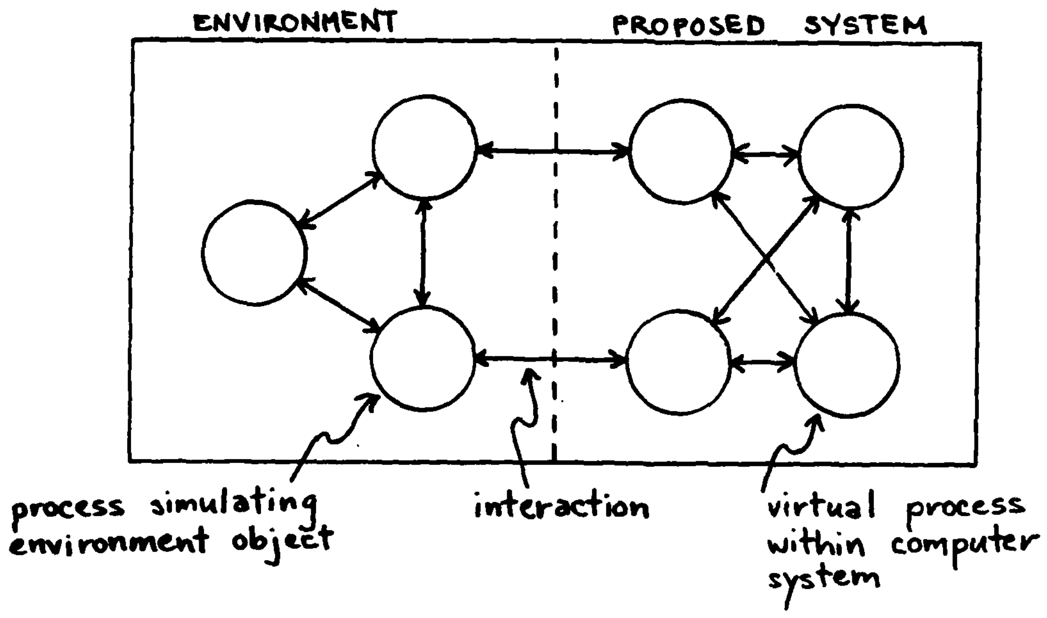
ENVIRONMENT
PROPOSED SYSTEM
process simulating environment object
interaction
virtual process within computer system

Figure 1. An operational requirements specification.

---

its environment. It is certainly valuable in specifying the system/environment boundary, because assumptions and protocols on both sides of it can be documented. Furthermore, many performance requirements are most naturally associated with the environment model.

(b) Both the system and environment are specified in terms of asynchronously interacting digital processes (although processes representing nondigital environment objects, such as people, are actually digital simulations of them). This formalism is general, abstract, and precise. It also captures directly the concurrency and synchronization which are crucial to embedded systems.

(c) The specification is executable. This makes it possible to debug and validate requirements specifications by testing them, especially by giving demonstrations to potential users. It also leads to a ready-made test bed, performance simulations if necessary, a concrete standard for acceptance testing, and the strongest possible notion of internal consistency.

(d) Computations within each process are specified using an applicative notation, i.e. one based on side-effect-free evaluation of expressions. Applicative languages have tremendous powers of abstraction, and applicative programming is the epitome of the top-down style.

These features are embodied in the language PAISLey (Process-oriented, Applicative, Interpretable Specification Language), which has been used "on paper" to specify a wide variety of embedded system requirements. An implementation is now being planned. The state of the project is best summarized in [Zave 80]; [Zave & Yeh 81] contains an extended example.

1.2. A Preview

The examples contained herein all come from the domain of process-control systems. The reason is simply that we had to start somewhere--and three interesting examples from this class presented themselves. ere is reason to believe, however, that process-control systems are a particularly representative and suggestive group. In [Zave 80] it is argued that "process control", defined as "providing continual feedback to an unintelligent environment", is the central concept behind all embedded systems. In other words, process-control systems are the quintessential embedded systems! Furthermore, the results seem quite generalizable to other application domains (see Section 5).

For the requirements analyst starting to build a model of a proposed system in its environment, the biggest problem is complexity. Section 2 outlines an approach to decomposing complexity which is compatible with user views of requirements and also supported by the PAISLey notation. Since this leads to two levels of requirements analysis, Sections 3 and 4 deal with those levels in turn.

2. THE BASIC APPROACH TO MANAGING COMPLEXITY

2.1. Incremental Development

The complexity of a large system must be managed by decomposing it, i.e. breaking the mass of decisions that must be made into reasonable chunks, each chunk being a subset of decisions which can be made in relative isolation from the others. When such a decomposition is available, a specification of the system can be developed incrementally, one chunk at a time; the history of the specification will be a sequence of versions, each formed by adding an increment to the previous version.

A useful technique for incremental development must provide a means, firmly grounded in the specification language being used, for isolating some decisions from others. In PAISLey, for instance, requirements decisions are validated by testing the specification. This implies that whatever technique we offer for incremental development in PAISLey should make it possible to execute each increment and version at the time it is developed.

A useful technique for incremental development must also provide guidelines for identifying appropriate increments. This is the methodological information that enables the analyst to make effective use of his specification tools.

2.2. A Functional Technique

Our technique for incremental development is based on the idea that a system performs a number of <u>functions</u> for its environment, and that requirements analysts and users naturally approach a system problem in terms of these functions. We base our increments on these functions, i.e. the PAISLey specification is developed one function at a time.

The reason that this works in PAISLey is that there exists an intuitive classification of system functions such that functions defined according to it have a one-to-one correspondence with processes in the PAISLey specification of the proposed system. The classification for process-control systems is given in Section 3. Furthermore, PAISLey is fully supportive of a mode of incremental development in which processes are being added, as described in Section 4.

The result is a two-step procedure for requirements analysis and specification. In the first step, the function/process structure of the system is determined and described informally. This step is not much different from the way that requirements analysis is usually done, except that the function classification improves completeness and consistency, and makes clear which parts of the environment belong in the operational specification.

In the second step, a PAISLey specification is developed one function/process at a time. The structure determined in the first step provides just enough information about system interconnections so that not-yet-specified functions can be adequately anticipated. The specification of each function entails a great deal of decision-making, so that handling them one at a time does decompose complexity substantially.

2.3. Processes in PAISLey

Since processes as they appear in PAISLey are central to this technique, we survey their major features. Formally, a process consists of a state space (set of all possible states) and a successor function (this "function" is of the mathematical variety) which computes each state from its predecessor. Thus a process goes through a sequence of process steps delineated by well-defined process states (Figure 2). The processes in a system perform in asynchronous parallel with each other, interacting as specified in their respective successor functions.

A process models a perpetual, cyclic activity, and its successor function specifies what it does during each cycle. The concept of cyclic behavior is a very natural one for embedded systems, and also correlates well with performance considerations. Most performance requirements seem to be expressable as constraints on the cycle times of individual processes in a PAISLey requirements specification.

PAISLey processes are also "distributed" processes, each one encapsulating its own local data and able to access the data of others only

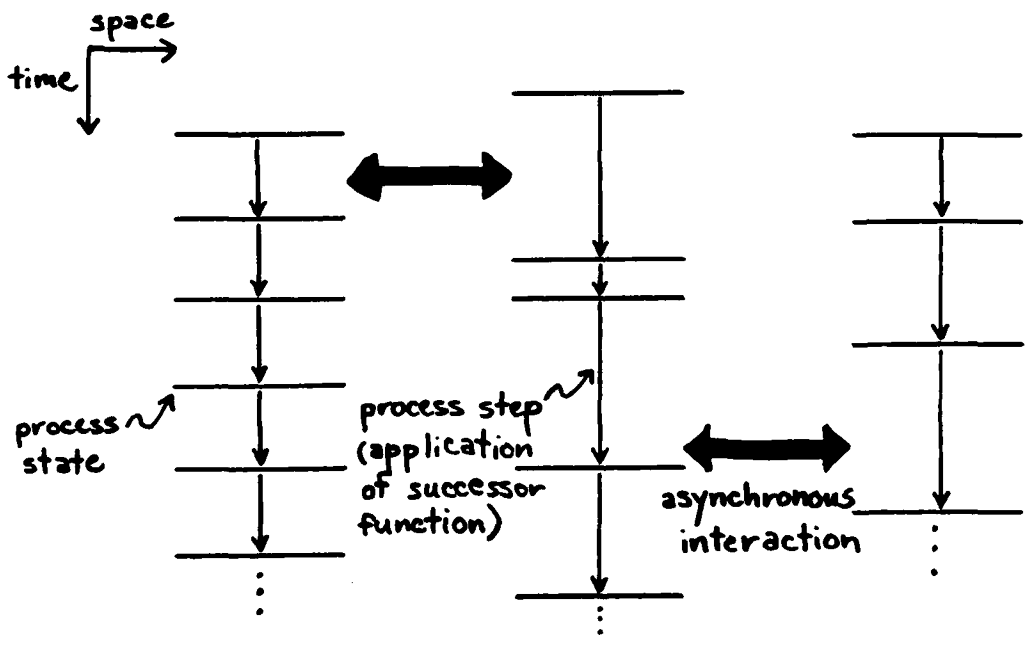

Figure 2. Processes in action.

through explicit, low-bandwidth communication. A database, for instance, would be specified as a process whose state contains the current data and whose successor function receives requests and honors them by accessing the data.

2.4. Some Justifications for the Functional Technique

This technique is perhaps unusual because the first line of defense against complexity is not abstraction (suppression of details) but partition (describing the whole in terms of its parts). Abstraction is not used until an individual function is being specified, at which time top-down elaboration can and should be used to provide another level of incremental structure.

This follows directly from the observation that, at the top level, a complex system performs a variety of functions which co-exist as equals (rather than some being subsidiaries of others). A business system, for instance, may fill orders, send monthly invoices, and re-order stock items when their quantities in the inventory are low. None of these functions could be described as a refinement of one of the others. The same philosophy is inherent in RSL ([Bell et al. 77], [Alford 77], [Davis & Vick 77]), in the domain of embedded systems: a top-level requirements specification in RSL consists of a set of parallel R-nets, each describing a separate function of the system.

Another way to see that functions must be differentiated is to consider time. The reason that sending invoices and filling orders are separate functions of the business system, for instance, is that they are done at different times: invoices are sent monthly (and thus can refer to more than one order). If an invoice were sent for each order, on the other hand, invoicing might be considered a subfunction of filling orders. In PAISLey each process has its own natural cycle time, which corresponds to the period of the function it carries out.

3. THE FUNCTION/PROCESS STRUCTURE OF PROCESS-CONTROL SYSTEMS

3.1. The Five Types of Process-Control Functions

A process-control system performs functions for its environment--controlling inanimate objects and providing information to animate ones--that are discernable as soon as a general solution to the problem posed in the statement of need is proposed. We will give a set of informal, but precise, rules for identifying the set of external functions performed by a process-control system, which fall into five categories.

A process-control system gets information about its environment through sensors, and reading a sensor is the first type of process-control function. Thus there is one "reader" process for each sensor, interacting both with the environment process modeling its sensor, and with all the system processes which use the values of that sensor. Its cycle time is the frequency at which the sensor must be read.

A reader process must maintain whatever data and perform whatever processing is needed to provide useful values to the rest of the system. If the sensor itself produces a value, for instance, then the reader process need do no more than read it at the specified intervals and offer it to the appropriate destinations. If the sensor just sends pulses, on the other hand, then the reader process must maintain an internal count of them.

The second type of process-control function is responding to a perceived condition in the environment. A "monitor" process carrying out such a function will interact with whatever reader processes it needs to detect the condition, and with whatever processes model or handle the recipients (actuators, human-readable output devices, alarms, etc.) of its response. It will store internally whatever information it needs to make history-sensitive decisions.

Since a monitor process receives the latest sensor information, checks for the condition, and effects its response all in one process step, its cycle time must have as an upper bound the response-time limit for its particular feedback loop. Thus one way to differentiate one function from another is disparate response-time limits. Other function differentiators are having different sensor inputs, having different effector outputs, or needing to keep different historical information. In practice it is not difficult to tell one function from another, because the factors that serve to define processes--such as performance, synchronization, and encapsulated data--also serve very well to differentiate them.

The third type of process-control function is providing information to other systems or to human users of the system. Information can be provided continuously (writing sampled data to a log tape), periodically (generating daily reports), or on demand (answering queries).

An "information" process interacts with whatever sensors it needs to get the information it deals in, and also with whatever processes request and/or receive the information it has to give. It stores in its state all the information it must "remember". Its cycle time is bounded by the response-time limit for providing information, and it is differentiated from other information functions by its time constraints, its interactions, and its data.

The fourth type of process-control function is reconfiguration, or making the system do something different. Individual functions can be turned on or off, or their parameters can be altered. A "reconfiguration" process orchestrates such transformations by receiving the stimuli for them (usually external requests) and translating these into instructions which update the control information of the affected functions/processes. Reconfiguration processes also enforce rules about who may reconfigure, what reconfigurations are valid at given times, etc.

The fifth type of process-control function is handling an input or output device. Inputs from an input device must be distributed to the functions for which they are intended. Outputs to an output device must be scheduled. In either case buffering, data conversion, etc. may be required. Device handling is different from sensor reading in that it is not a real-time function.

These types were derived from studying three systems in this class: (a) an industrial process-control system which adjusts coolant valves in response to temperature fluctuations, sounds an alarm when it detects a dangerous condition, answers queries about machine conditions from an operator, and prints reports on production and consumption of raw materials; (b) a data acquisition system which samples data and computes its statistical characteristics, and can write either the raw data or the statistics to several destinations (tape, printer, display); (c) the patient-monitoring system whose function/process structure is given in the next section. The types are surprisingly comprehensive, encompassing variations large and small.

3.2. Example: A Patient-Monitoring System

Figure 3 shows the function/process structure of a simple patient-monitoring system. We have given every type of function its own shape for nodes in the graph. In PAISLey specifications many processes are replicated, and this is indicated in process diagrams by drawing a double line around the

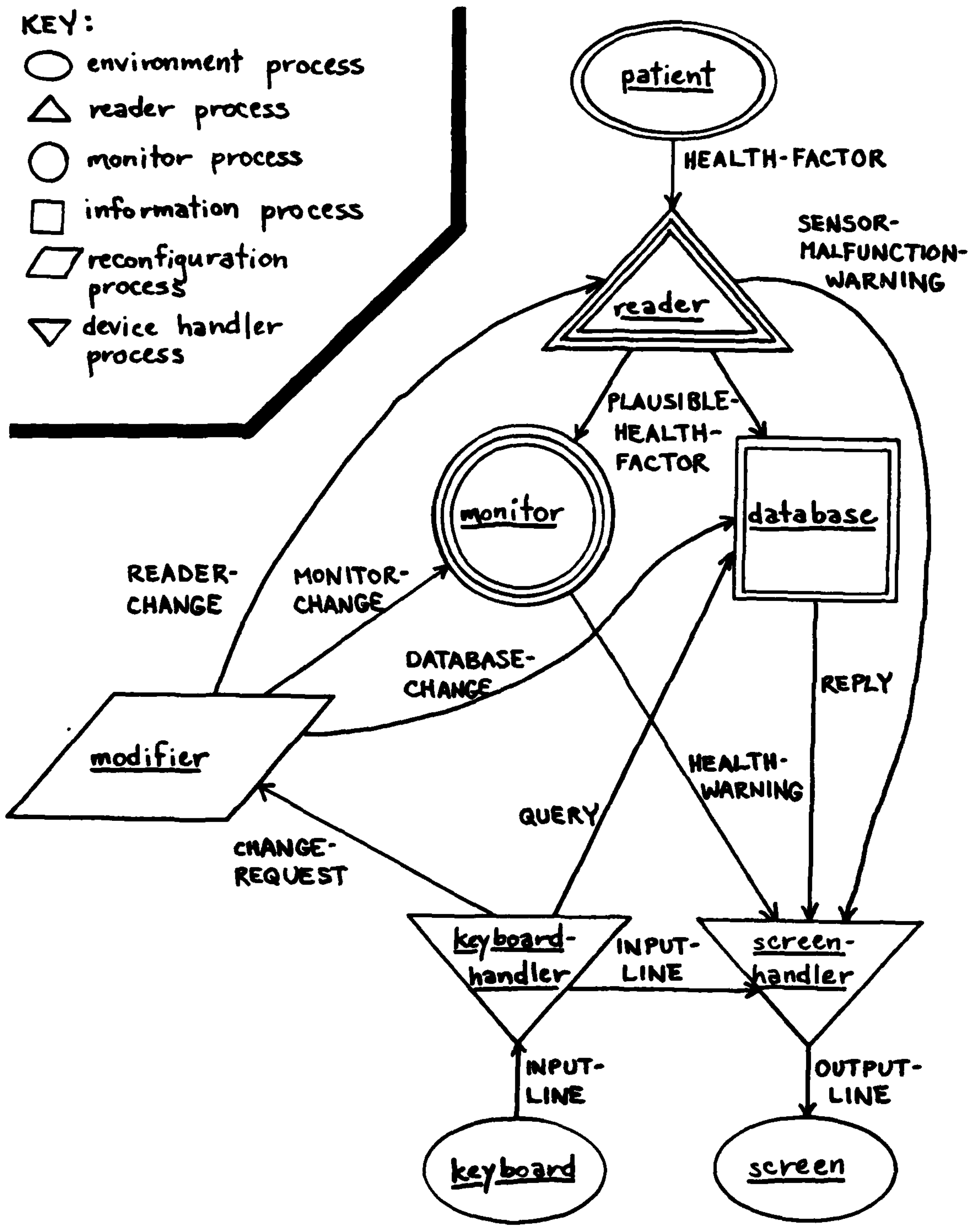

Figure 3. The function/process structure of a patient-monitoring system.

process. In Figure 3, for instance, there is a type of process in the environment modeling a patient. There is a nearly identical instance of this type of process (except for an index) for each patient in the intensive care unit.

There is a reader process for each sensor attached to each patient (triple lines indicate double indexing and replication). It reads its sensor at specified intervals. It also checks the reading against a standard of plausibility for that particular sensor, and generates a warning of sensor malfunction if the reading fails (note that it is *not* responding to a perceived condition in the environment, but to a perceived condition in the system's interface with its environment). Otherwise, the reading is passed on to several destinations within the system.

There is also a monitor process for each sensor of each patient. It checks each factor for safety, and generates a warning if anything is amiss.

There is a database process (a type of information process) for each patient. This process stores factors read from its patient, and answers queries concerning the patient's history.

There is one reconfiguration process. Although this system undergoes no major reconfigurations, many of its functions are controlled by parameters which can be changed by doctors and nurses. The frequency with which a particular sensor is read, for instance, is a changeable parameter (a value stored in its state) of a reader process.

Finally, there are two handler processes to manage the two I/O devices of this system, the keyboard and the screen of a full-duplex terminal. The screen handler must arrange for the sharing of the screen among patient histories, echoes, and warnings, in order of ascending priority and descending length. This will entail internal buffering of patient histories, so that they can be interrupted on a line-by-line basis by warnings.

In Figure 3 process interactions are labeled with the type of data transferred. This makes the function/process diagram essentially equivalent to a "dataflow" or data-access diagram, which has long been the mainstay of requirements analysis for all types of system ([Ross 77], [Teichroew & Hershey 77], [Miyamoto & Yeh 81]). This shows that PAISLey specification techniques need not be inferior to conventional ones for intuition and communication, even though the PAISLey view of the proposed system has the potential for being developed into a formal, executable, and analyzable specification at any level of detail.

3.3. Further Comments

The process structures derived according to this technique are not compromises or merely satisfactory solutions--they are the best process structures we know how to invent, taking into account both logical and performance requirements. This conclusion was reached by comparing process structures derived according to these rules with process structures for the same systems that were developed before the rules were formulated.

One indication of the quality of the proposed process structures is that they seem to be easily modifiable. Adding, deleting, and modifying functions are local operations, involving mainly the function/process itself. At most, other processes would have to undergo minor modifications to provide output to new destinations or accept input from new sources. In other words, if it proves true that changes to requirements, as well as requirements, are understood in terms of functions, then a modularization based on functions will be modifiable as well as intuitive.

In the interest of modifiability, we should probably avoid "optimizations" such as coalescing functions/processes which are very closely coupled (such as corresponding readers and monitors in the patient-monitoring system, each of which takes exactly one step per health factor read). A modification that caused a monitor to get input from more than one sensor, or caused a reader to pass its data to more than one monitor, would loosen the coupling considerably.

The better we understand the five types of function, the more stereotyped they will seem. It may even be possible to provide a "skeleton" specification for a type of function, with an enumeration of the decisions to be made, and places to specify the results of those decisions. Even the functional structure as currently understood is a force for completeness in requirements analysis, as it generates a list of questions the analyst must answer about each function (see 4.1).

4. ELABORATING THE FUNCTIONS

4.1. Intra-Function Decisions

After the process structure is determined, the specification is developed by elaborating one function/process at a time (environment processes can be developed with their associated readers and handlers). This provides substantial decomposition of complexity, because within the boundaries of each function there are still a large number of decisions to be made. This will be illustrated by enumerating some of the decisions associated with each function of the patient-monitoring system.

Reader processes are probably the simplest of all five types in the patient-monitoring system, because it is assumed that the sensing devices attached to patients deliver continuous usable values. Nevertheless, it must still be determined whether sensors (and their readers) can be turned off and on, how sensor malfunction can be detected, and with what frequency each sensor is to be read (Is the frequency fixed or variable? If fixed, is it different for each type of sensor? If variable, is there a default?).

A monitor process checks factors and emits warnings, but does it check each reading (or only occasional readings, or batches of them)? If it checks each reading, does it emit a warning for each bad one (even though this could flood the screen)? What do the warnings contain? How are dangerous conditions detected? Does detection involve a history of the past several readings? Does it involve variable parameters?

From the process structure we know only that the information (database) process for each patient keeps health factors read from the patient, and answers queries about them. Are all the factors kept? Are they associated with timestamps? How is old information purged to keep the database of manageable size? What are the valid queries, and what are the replies to them?

In the patient-monitoring system there are no major reconfigurations, only alterations of parameters. Thus in elaborating the reconfiguration function it is only necessary to collect all the variable parameters and their potential ranges, and decide on a convenient language in which users can describe changes or sets of changes. The specification will be completed by showing the act of reconfiguration, i.e. the sending of instructions from the reconfiguration process to the other processes causing them to update their states.

Although specification of the keyboard handler demands few new decisions, the screen handler is another matter. How exactly is the screen shared among the various functions that use it? How much buffering is needed, based on performance requirements, to keep the system running smoothly?

It is worth noting in passing that elaboration of a function may entail the introduction of additional processes, but these new processes will be entirely subsidiary to the functions/processes from which they came, i.e. they will not interact with any other processes in the requirements specification. An example would be a "timer" process, used to notify a reader process that it was time for the next reading.

4.2. The Connections Between Processes

Despite the relative independence of functions, they must surely interact. Although the general pattern of interaction is known from the function/process structure, how can processes be specified and tested when other processes with which they interact are missing? This aspect of incremental development is supported by the structures for process interaction in PAISLey.

In PAISLey interactions are specified within the applicative expression specifying the successor function of a process (in the rest of this section we are talking about mathematical functions again). An asynchronous interaction site is first (as the expression is being elaborated) represented as a primitive "pseudo-function". All interactions involve mutual synchronization and data transfer between two sites.

For instance, in a case where one process must send a message to another, the sender might use a pseudo-function

send-message: MESSAGE ---> ACKNOWLEDGMENT

to do it (this declaration says that the pseudo-function "send-message" is a mapping from the set "MESSAGE" to the set "ACKNOWLEDGMENT"). The recipient might use

receive-message: ---> MESSAGE,

which is a pseudo-function with no argument. We describe these as "pseudo-functions" because they are mappings which get evaluated during the course of evaluation of their containing expressions. They are not true functions, however, because their values do not depend on their arguments--rather, their values depend on the processes with which they interact.

In the final specification "send-message" and "receive-message" will be defined in terms of PAISLey primitives which actually do the interacting. In the meantime, a primitive (undefined) pseudo-function such as "receive-message" can be evaluated during execution of the specification simply by choosing a value at random from the set "MESSAGE", or possibly asking a user at a terminal to supply a value.

Thus the scenario for incremental development is as follows (Figure 4): (a) When the current version of the specification contains one process which is a party to a particular interaction, but does not contain both parties, the interaction site is left as a primitive pseudo-function. During execution, its values are chosen randomly, by a user, or by default. (b) When the increment containing the process which is the other party to the interaction is being prepared, it can be tested in isolation by leaving all of its interaction sites as primitive pseudo-functions, and handling them during execution as above. (c) When the current version and the new increment are integrated, the interactions which can now be completely specified, because both processes are present, are specified and tested. This is done by elaborating both interaction sites in terms of PAISLey interaction primitives.

Since interactions are symmetric at this level of abstraction, the scenario places no restrictions whatsoever on the order in which increments

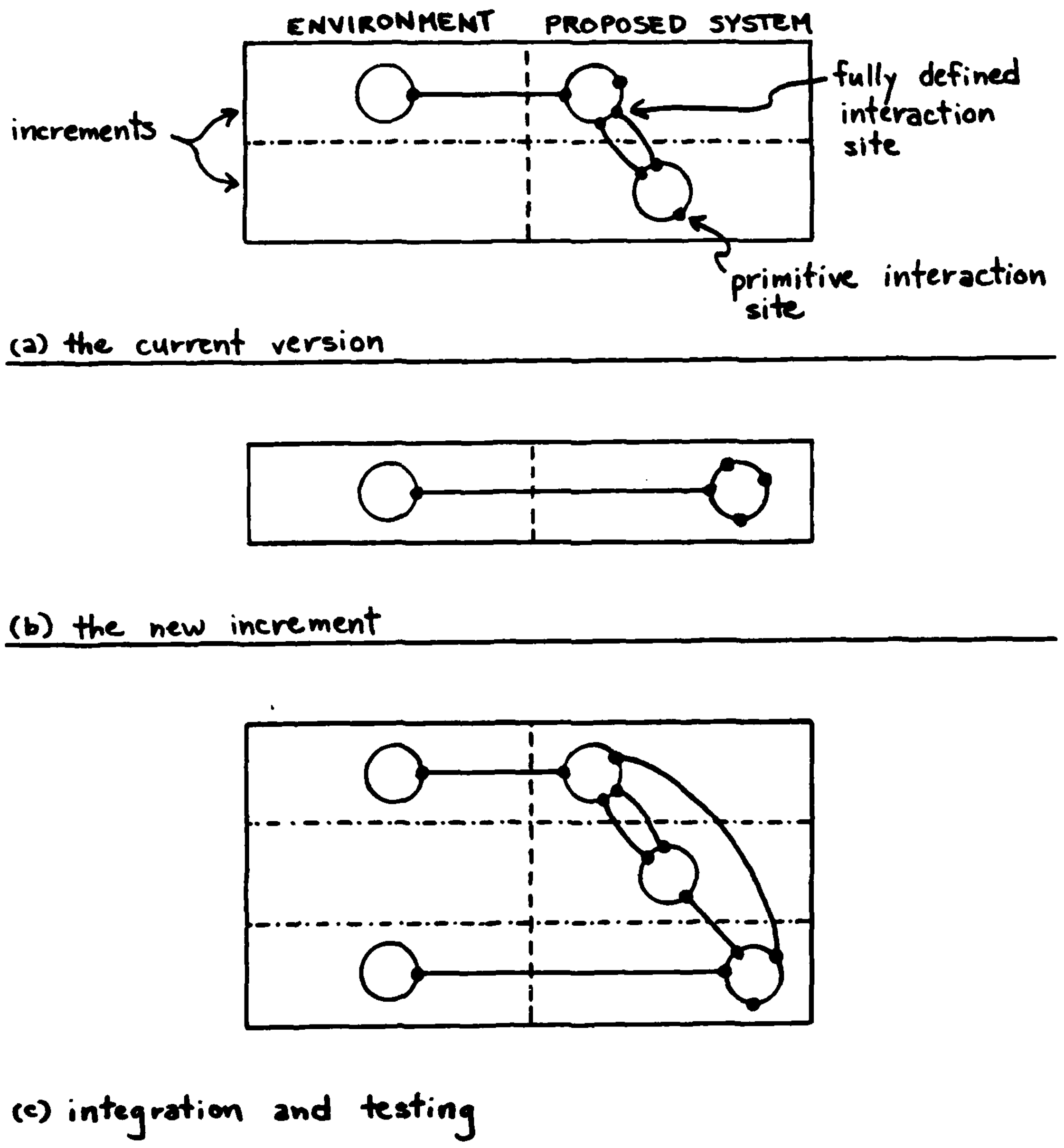
ENVIRONMENT
PROPOSED SYSTEM
increments
fully defined interaction site
primitive interaction site
(a) the current version
(b) the new increment
(c) integration and testing

Figure 4. The scenario for incremental development.

are developed. Perhaps more experience will reveal that certain types of function are better developed before other types, but so far nothing conclusive has come up. In the meantime requirements analysts can be guided by their own judgment, without hindrance from the specification language.

5. EXTENSION TO OTHER APPLICATION DOMAINS

It seems likely that this functional viewpoint on formal requirements will generalize to other types of system, demanding only a handful of new function types for each new domain. In fact, several of the function types can be used for any application, as follows:

Another way to categorize the functions is as (a) "outer" functions (readers, handlers) which manage the interface between the system and objects of the environment, (b) "inner" functions (monitors, *information functions*) which correspond most closely to the purpose of the system, and (c) reconfiguration functions. Outer functions and reconfigurations will be much the same in any application domain; only the types of inner function will vary.

The inner functions of process-control systems do have a special property, however, which makes them so easy to separate. Each function <u>reacts</u> to sensor data, but never <u>changes</u> it. If the state of the environment changes because of actions of the process-control system, then that change will eventually be reflected in different sensor values coming in.

As a result, each inner function/process is free to receive and store whatever sensor data it needs to carry out its function, even though there is considerable overlap between its working data and that of some other function. Consistency is not a problem because (semantically) the data is not changed by either function. Redundancy is not a problem because this is a requirements specification, not a design--resource usage is not yet an issue. This is what made it possible for us to concentrate analysis on functions, knowing that whatever data was needed could be supplied without complications.

For a database system the situation will be somewhat different. A system which maintains an internal image of an inventory, for instance, does not get its knowledge of the inventory from sensors. Rather, it constructs its image of the inventory from knowledge of the transactions that add or subtract inventory items. Thus internal system functions actually change the system's image of its environment, and data shared among functions cannot exist redundantly without causing consistency problems.

The advantages of the process-control structure can probably be adapted to this situation simply by encapsulating units of shared, modifiable data in their own functions/processes (analogous to data modules in the data abstraction literature). These processes can take the place of reader processes in supplying unambiguous real-world information to all the other functions in the system. Nevertheless, the need to deal with redundancy and consistency of computations and data, at the requirements level, raises many intriguing questions about the abstract structure of computer systems.

6. CONCLUSION

This paper has described a technique by which system requirements can be analyzed and specified formally, starting from intuitive functions in the initial proposal. The emphasis has been on decomposing complexity, in response to the major problem confronting the developers of real systems. Although these are preliminary results, and still need refinement and experience, they suggest that we can be optimistic about finding and exploiting regular, intelligible structure in both functional requirements and

PAISLey specifications.

REFERENCES

[Alford 77]
Mack W. Alford, "A Requirements Engineering Methodology for Real-Time Processing Requirements", IEEE Trans. Software Engr. SE-3, January 1977, pp. 60-69.

[Bell et al. 77]
Thomas E. Bell, David C. Bixler, and Margaret E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering", IEEE Trans. Software Engr. 3, January 1977, pp. 49-60.

[Davis & Vick 77]
Carl G. Davis and Charles R. Vick, "The Software Development System", IEEE Trans. Software Engr. SE-3, January 1977, pp. 69-84.

[Miyamoto & Yeh 81]
Isao Miyamoto and Raymond T. Yeh, "A Software Requirements Analysis and Definition Methodology for Business Data Processing", Proc. NCC, Chicago, Ill., May 1981, pp. 571-581.

[Ross 77]
Douglas T. Ross, "Structured Analysis (SA): A Language for Communicating Ideas", IEEE Trans. Software Engr. SE-3, January 1977, pp. 16-34.

[Teichroew & Hershey 77]
Daniel Teichroew and Ernest A. Hershey III, "PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems", IEEE Trans. Software Engr. SE-3, January 1977, pp. 41-48.

[Zave 80]
Pamela Zave, "The Operational Approach to Requirements Specification for Embedded Systems", University of Maryland Computer Science TR-976, December 1980.

[Zave & Yeh 81]
Pamela Zave and Raymond T. Yeh, "Executable Requirements for Embedded Systems", Proc. Fifth International Conference on Software Engineering, San Diego, Cal., March 1981, pp. 295-304.